

Programación en Lenguaje Java

Tema 8. Herencia y polimorfismo



Michael González Harbour
Mario Aldea Rivas

Departamento de Matemáticas,
Estadística y Computación

Este tema se publica bajo Licencia:

[Creative Commons BY-NC-SA 4.0](https://creativecommons.org/licenses/by-nc-sa/4.0/)

Programación en Java

1. Introducción a los lenguajes de programación
2. Datos y expresiones
3. Estructuras algorítmicas
4. Datos Compuestos
5. Entrada/salida
6. Clases, referencias y objetos
7. Modularidad y abstracción
- 8. Herencia y polimorfismo**
 - Herencia. Clases abstractas. Polimorfismo. La clase Object
9. Tratamiento de errores
10. Entrada/salida con ficheros
11. Pruebas

8.1 Herencia

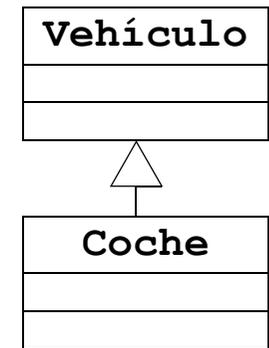
Relación de herencia:

- *todos los* coches *son* vehículos

La herencia es un mecanismo por el que se pueden crear nuevas clases a partir de otras existentes,

- heredando, y posiblemente modificando, y/o añadiendo operaciones
- heredando y posiblemente añadiendo atributos

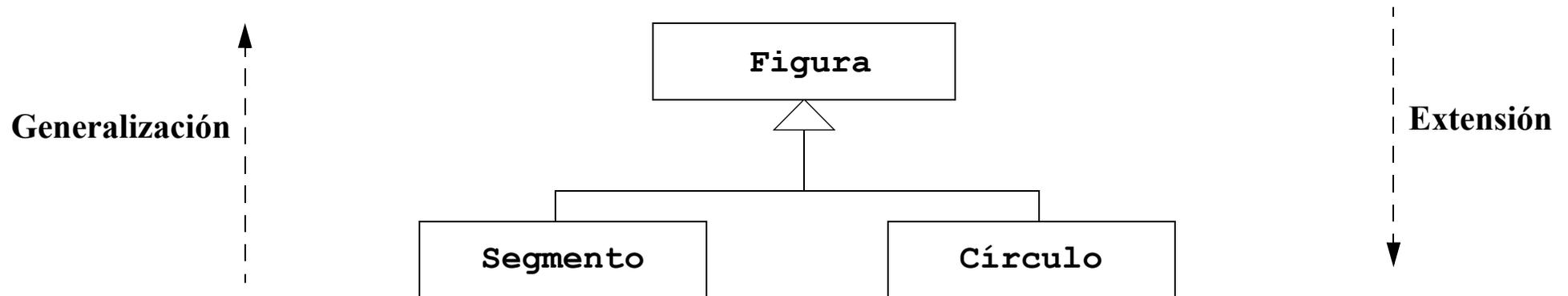
Observar que una operación o atributo no puede ser suprimido en el mecanismo de herencia



Nomenclatura

clase original	superclase	padre	Vehículo
clase extendida	subclase	hijo	Coche

La **Herencia** también se denomina **Extensión** o **Generalización**



La **Herencia** (y el Polimorfismo) son unos de los **conceptos más importantes y diferenciadores** de la Programación Orientada a Objetos

Herencia de operaciones

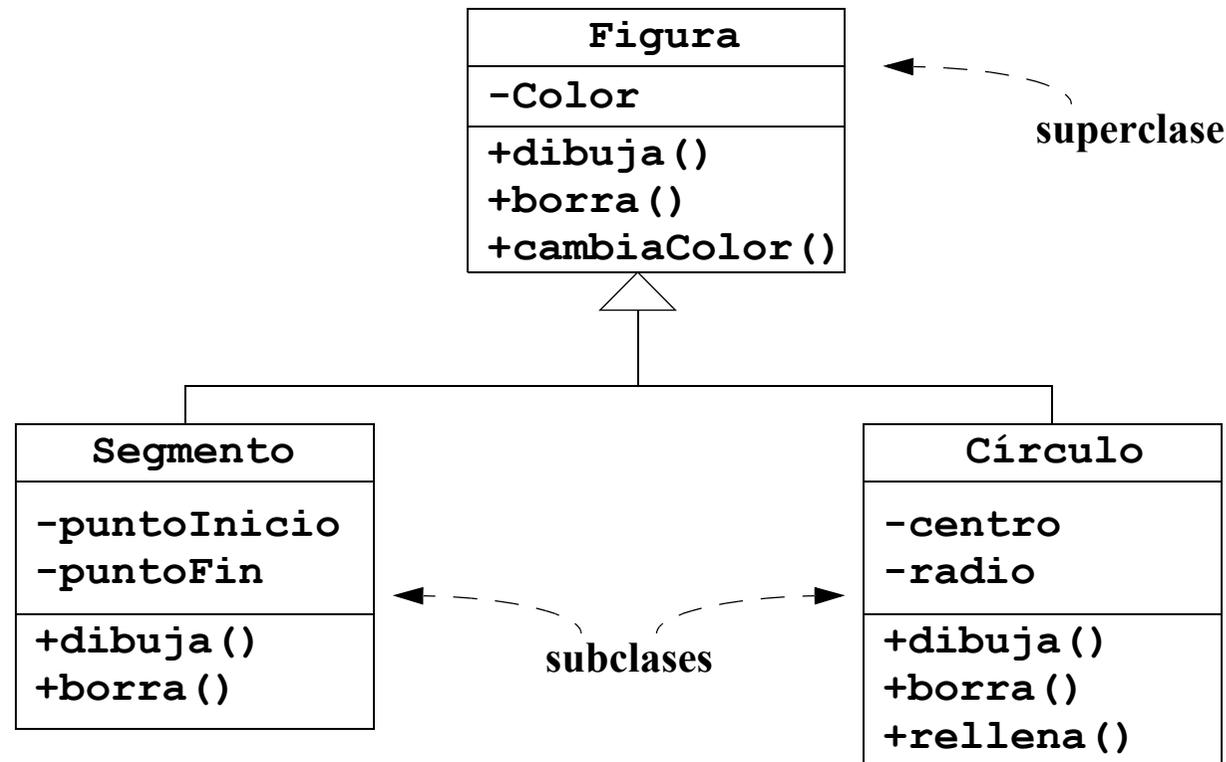
Al extender una clase

- se **heredan** todas las operaciones del padre
- se puede **añadir** nuevas operaciones

La subclase puede elegir para las operaciones heredadas:

- **redefinir** la operación: se vuelve a escribir
 - la nueva operación puede usar la del padre y hacer más cosas
 - o puede ser totalmente diferente
- no hacer nada y heredarla tal como está en el padre

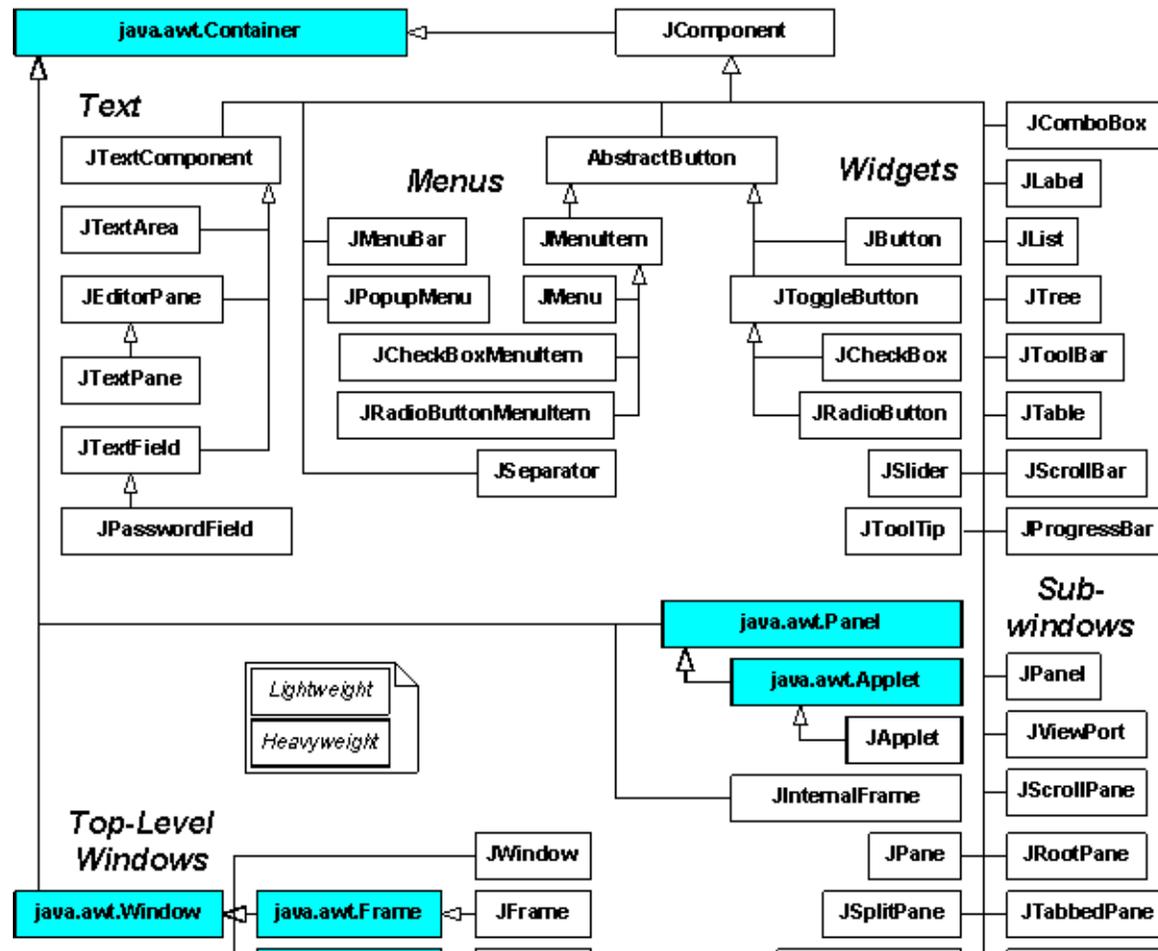
Herencia en un diagrama de clases



- Los atributos y métodos de la superclase no se repiten en las subclases
 - salvo que se hayan redefinido

Jerarquías de clases

La herencia puede aplicarse en sucesivos niveles, creando grandes *jerarquías de clases*



Jerarquía de clases de la biblioteca gráfica Swing

Ventajas y desventajas del uso de la herencia

Ventajas:

- +Mejora el **diseño**

Permite modelar relaciones de tipo "es un" que se dan en los problemas que se pretenden resolver

- +Permite la **reutilización** del código

Los métodos de la clase padre se reutilizan en las clases hijas

- +Facilita la **extensión** de las aplicaciones

Añadir una nueva subclase no requiere modificar ninguna otra clase de nuestro diseño

Principal desventaja:

- Aumenta el **acoplamiento**

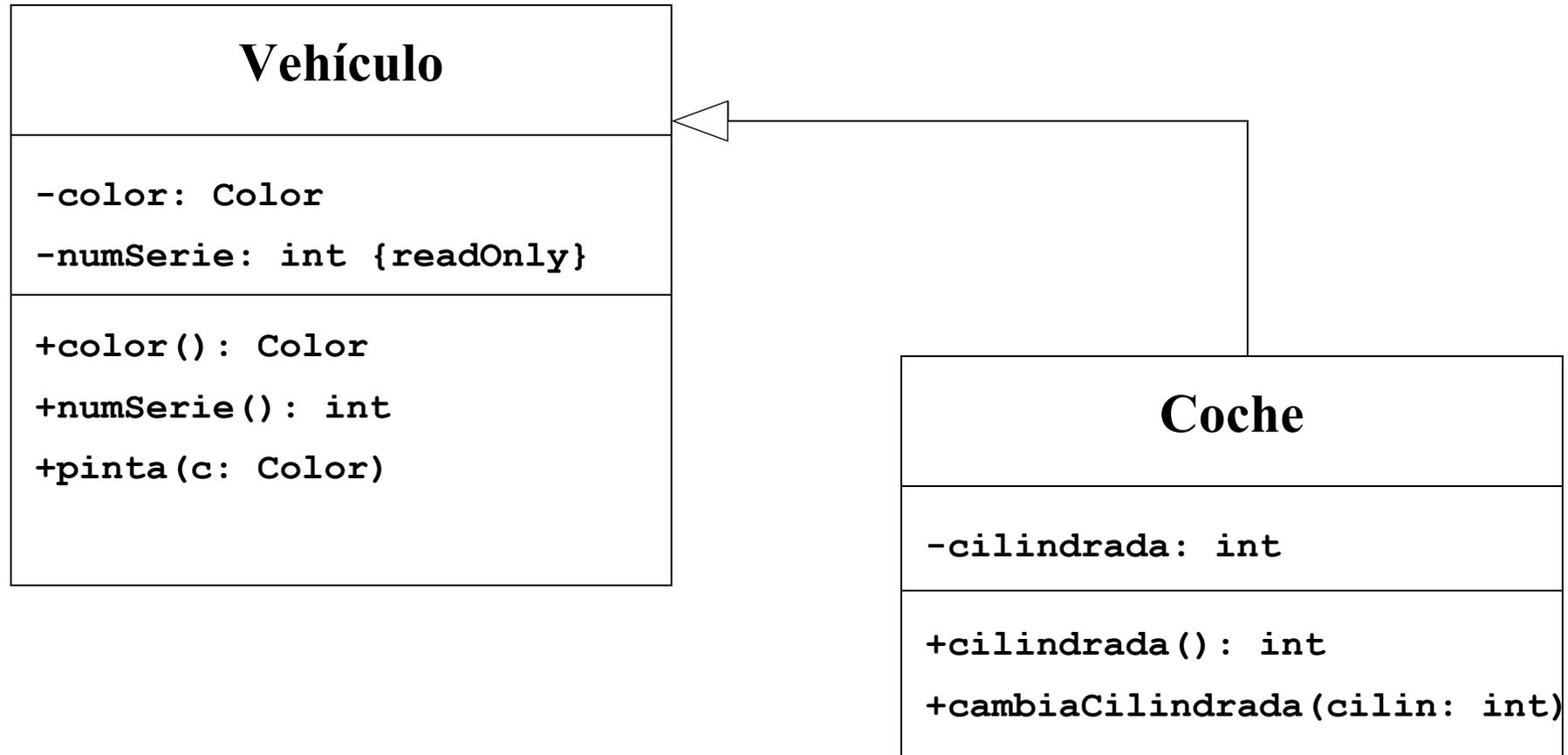
Las subclases están íntimamente acopladas con la superclase

Ejemplo sencillo

Clase que representa un vehículo cualquiera

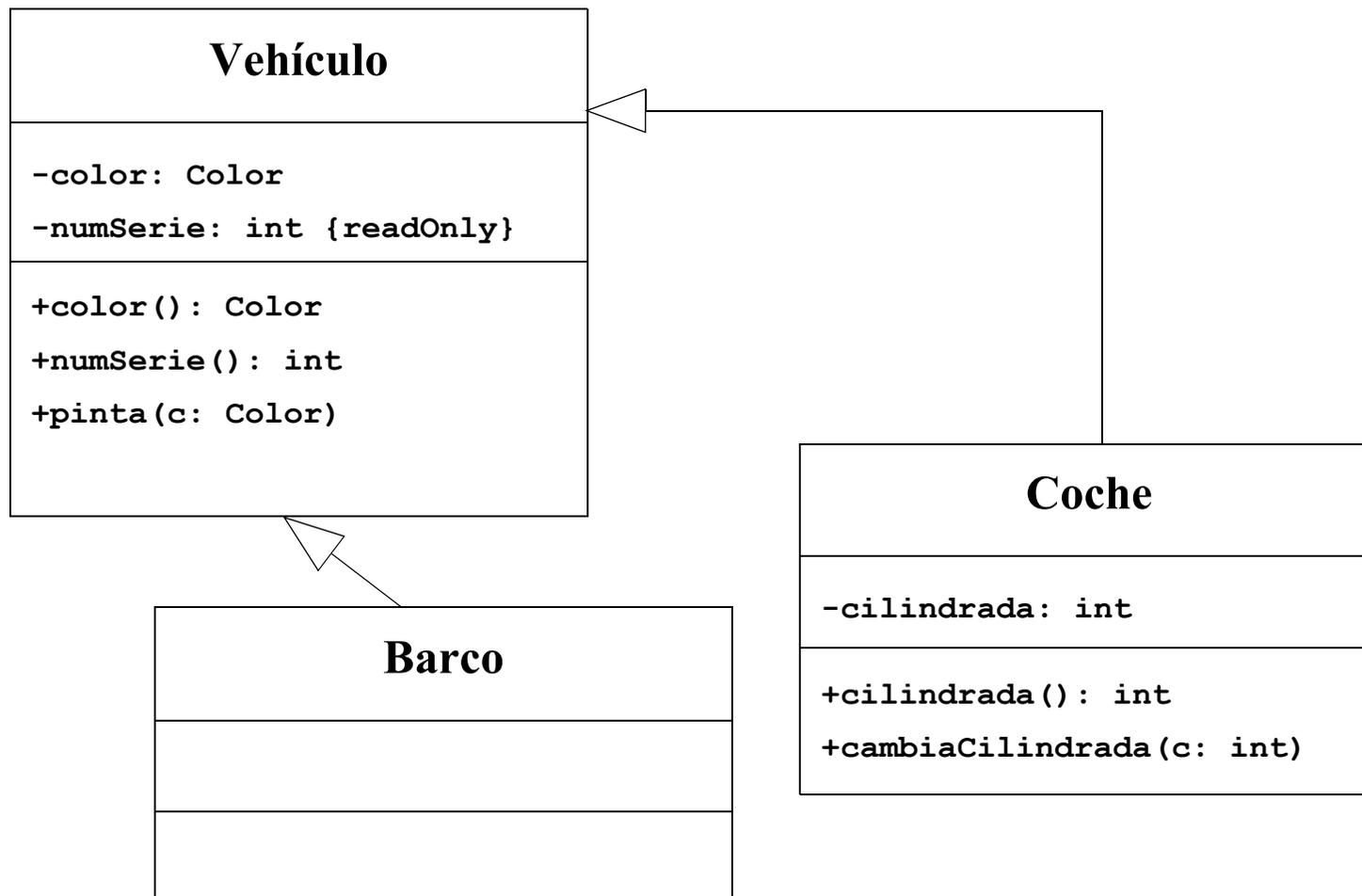
Vehículo
<code>-color: Color</code> <code>-numSerie: int {readOnly}</code>
<code>+color(): Color</code> <code>+numSerie(): int</code> <code>+pinta(c: Color)</code>

Ejemplo sencillo: subclase Coche



- La clase **Coche** añade el atributo `cilindrada` y los métodos para gestionar dicho atributo

Ejemplo sencillo: subclase Barco



(Observar que se puede extender una clase sin añadir atributos ni métodos)

Implementación del ejemplo en Java

```
/**
 * Clase que representa un vehículo cualquiera
 */
public class Vehículo
{
    // colores de los que se puede pintar un vehículo
    public static enum Color {ROJO, VERDE, AZUL}

    // atributos
    private Color color;
    private final int numSerie;

    /**
     * Construye un vehículo
     * @param color color del vehículo
     * @param numSerie número de serie del vehículo
     */
    public Vehículo(Color color, int numSerie)
    {
        this.color=color;
        this.numSerie=numSerie;
    }
}
```

```
    /**
 * Retorna el color del vehículo
 * @return color del vehículo
 */
public Color color()
{
    return color;
}
/**
 * Retorna el numero de serie del vehículo
 * @return numero de serie del vehículo
 */
public int numSerie()
{
    return numSerie;
}

/**
 * Pinta el vehículo de un color
 * @param nuevoColor color con el que pintar el vehículo
 */
public void pinta(Color c)
{
    color = c;
}
}
```

```
public class Coche extends Vehículo
{
    // cilindrada del coche
    private int cilindrada;

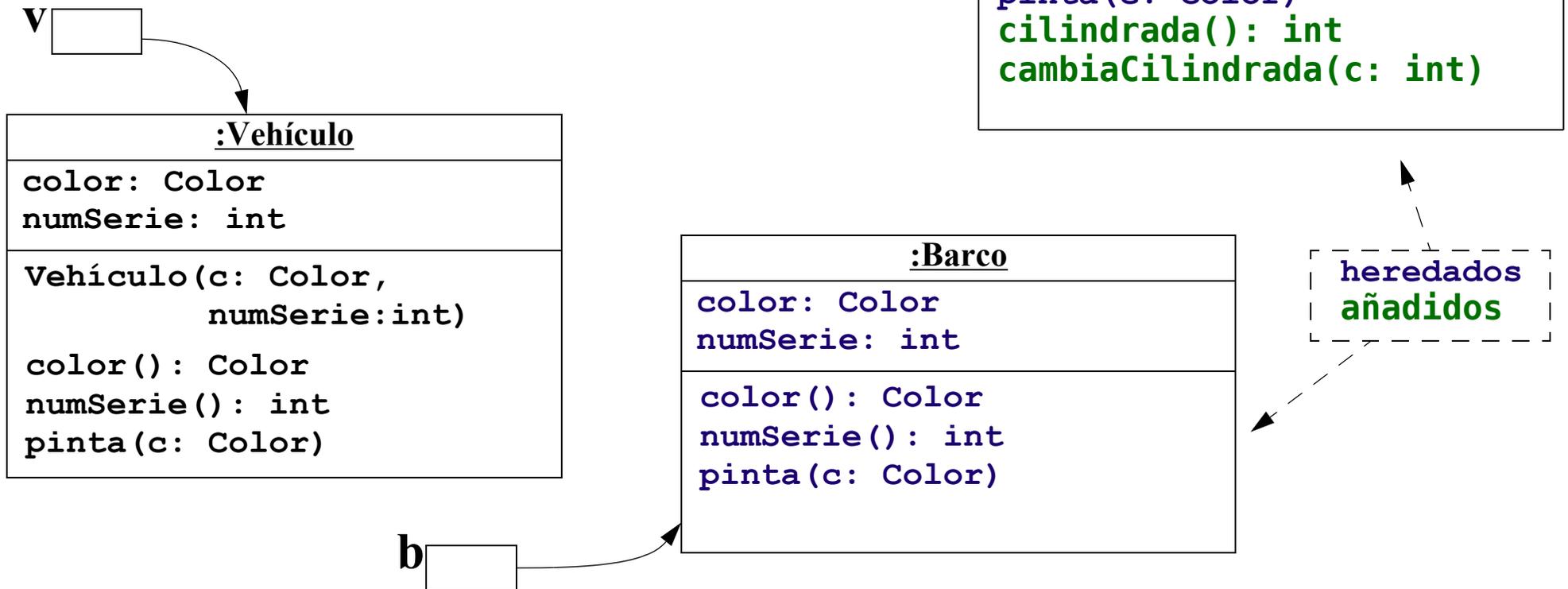
    /** Retorna la cilindrada del coche ... */
    public int cilindrada(){
        return cilindrada;
    }

    /** Cambia la cilindrada del coche ... */
    public void cambiaCilindrada(int c) {
        this.cilindrada=c;
    }
}
```

```
public class Barco extends Vehículo  
{  
  
}
```

Ejemplo: objetos y herencia

```
Vehículo v = new Vehículo();  
Coche c = new Coche();  
Barco b = new Barco();
```



Herencia y Constructores en Java

Los constructores no se heredan

- las subclases deben definir su propio constructor

Normalmente será necesario inicializar los atributos de la superclase

- para ello se llama a su constructor desde el de la subclase

```
/** constructor de una subclase */  
public Subclase(parámetros...) {  
    // invoca el constructor de la superclase  
    super(parámetros para la superclase);  
    // inicializa sus atributos  
    . . .  
}
```

- la llamada a “**super**” debe ser la primera instrucción del constructor de la subclase

Si desde un constructor de una subclase no se llama expresamente al de la superclase

- el compilador añade la llamada al constructor sin parámetros

```
public Subclase(int i){  
    this.i=i;  
}
```

se convierte en

```
public Subclase(int i){  
    super();  
    this.i=i;  
}
```

- en el caso de que la superclase no tenga un constructor sin parámetros se produciría un error de compilación

Redefiniendo operaciones

Una subclase puede redefinir ("*override*") una operación en lugar de heredarla directamente

Es conveniente indicarlo utilizando la anotación **@Override**

- para informar al compilador de que el método redefine uno de la superclase
- y detectar el error en caso de que no lo haga

Ejemplo: redefinición errónea del método `toString()`

```
@Override  
public String toString()  
    ...  
}
```

Gracias a la anotación **@Override**, el compilador nos informa de que `toString()` NO redefine ningún método de la superclase

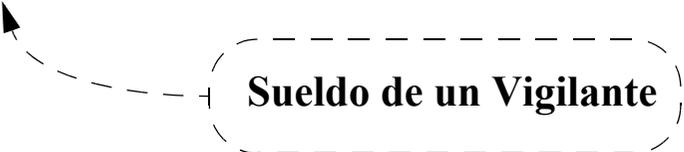
Invocando operaciones de la superclase

En muchas ocasiones (no siempre) la operación redefinida invoca la de la superclase

- se usa para ello la palabra reservada **super**
`super.nombreMétodo(parametros...);`
- se refiere a la superclase del objeto actual

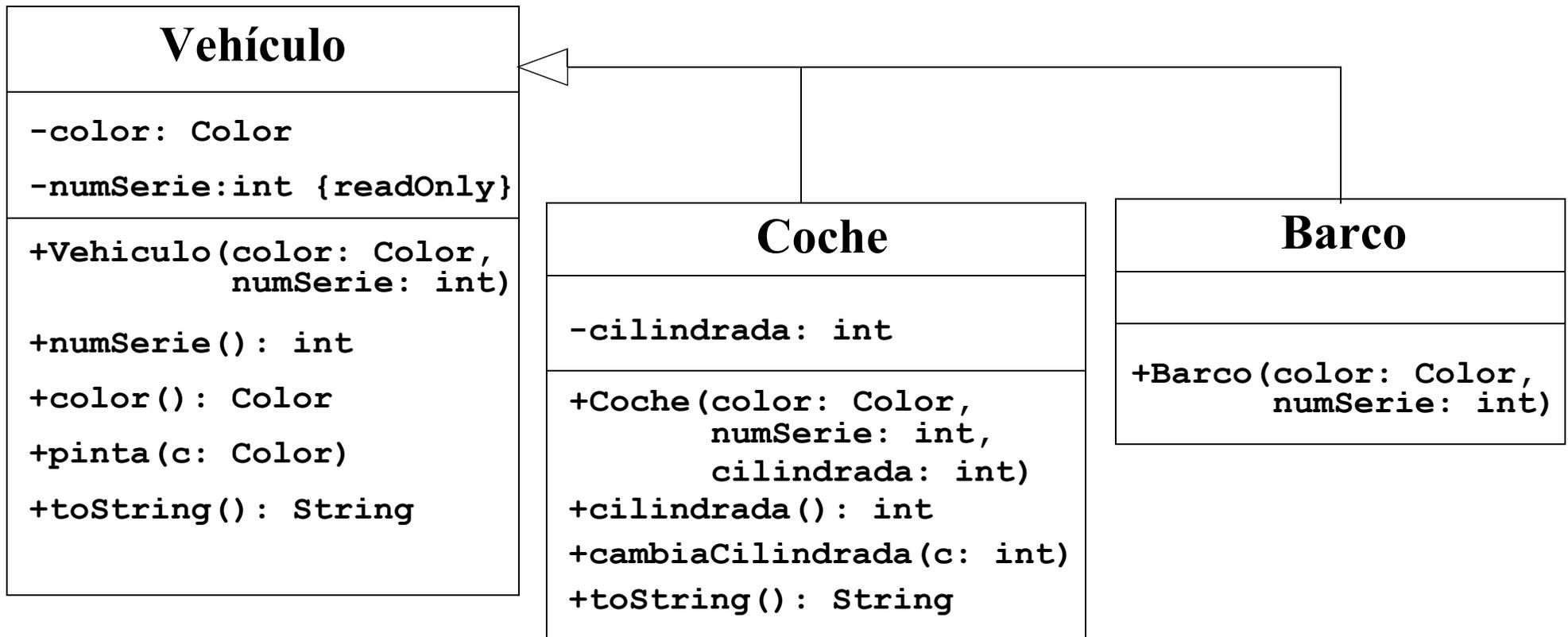
Ejemplo:

```
public class VigilanteNocturno extends Vigilante {  
    ...  
  
    @Override  
    public float sueldo() {  
        return super.sueldo() + PLUS_NOCTURNIDAD;  
    }  
}
```



Ejemplo: constructores y redefinición de operaciones

Modificamos las clases para añadir un constructor y un método que retorna en un String los datos del objeto



Ejemplo: clase Vehículo

```
public class Vehículo {
    // colores de los que se puede pintar un vehículo
    public static enum Color {ROJO, VERDE, AZUL}
    // atributos privados
    private Color color;
    private final int numSerie;

    /**
     * Construye un vehículo
     * @param color color del vehículo
     * @param numSerie número de serie del vehículo
     */
    public Vehículo(Color color, int numSerie) {
        this.color = color;
        this.numSerie = numSerie;
    }
}
```

```

public int numSerie() {...}
public Color color() {...}
public void pinta(Color c) {...}

```

No repetimos el código
(es igual que en el ejemplo anterior)

```

/**
 * Retorna un texto con los datos del vehículo
 * @return texto con los datos del vehículo
 */
@Override
public String toString() {
    return "Vehículo -> numSerie= " +
        numSerie + ", color= " + color;
}
}

```

Ejemplo: subclase Coche

```
public class Coche extends Vehículo {  
  
    // cilindrada del coche  
    private int cilindrada;  
  
    /**  
     * Construye un coche  
     * @param color color del coche  
     * @param numSerie número de serie del coche  
     * @param cilindrada cilindrada del coche  
     */  
    public Coche(Color color, int numSerie,  
                 int cilindrada) {  
        super(color, numSerie);  
        this.cilindrada = cilindrada;  
    }  
}
```

```
/** Obtiene la cilindrada del coche ... */  
public int cilindrada() {  
    return cilindrada;  
}  
  
/** Cambia la cilindrada del coche ... */  
public void cambiaCilindrada(int nueva) {  
    cilindrada = nueva;  
}  
  
@Override  
public String toString() {  
    return super.toString() + ", cilindrada= "  
        + cilindrada;  
}  
}
```

Ejemplo: subclase Barco

```
public class Barco extends Vehículo {  
  
    /**  
     * Construye un barco  
     * @param color color del barco  
     * @param numSerie número de serie del barco  
     */  
    public Barco(Color color, int numSerie) {  
        super(color, numSerie);  
    }  
  
}
```

Modificador de acceso `protected`

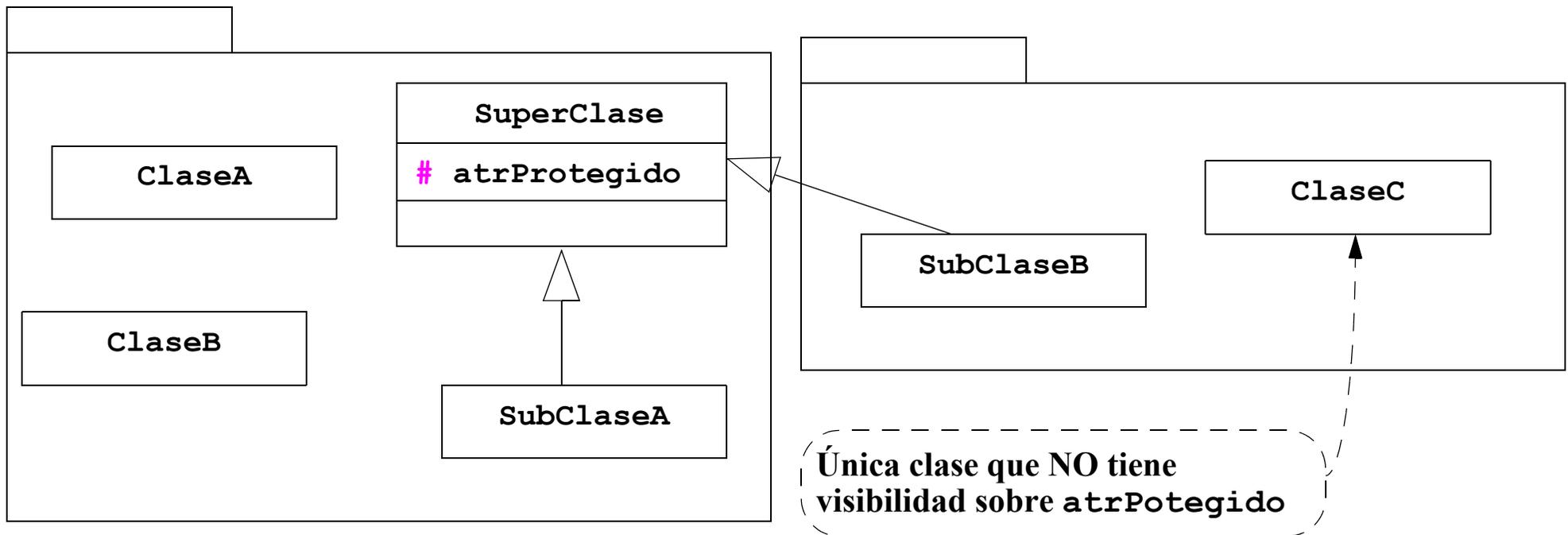
Modificadores de acceso para miembros de clases:

- `<ninguno>`: accesible desde el paquete
- `public`: accesible desde todo el programa
- `private`: accesible sólo desde esa clase
- `protected`: accesible desde sus subclases y, en Java, desde cualquier clase en el mismo paquete

UnaClase
+ <code>atrPúblico</code>
- <code>atrPrivado</code>
~ <code>atrPaquete</code>
<code>atrProtegido</code>
+ <code>metPúblico</code>
- <code>metPrivado</code>
~ <code>metPaquete</code>
<code>metProtegido</code>

En general, definir ***atributos protected en Java NO es una buena práctica*** de programación

- ese atributo sería accesible desde cualquier subclase
 - puede haber muchas y eso complicaría enormemente la tarea de mantenimiento
- además (en Java) el atributo es accesible desde todas las clases del paquete (subclases o no)



8.2 Clases abstractas

En ocasiones definimos clases de las que no pretendemos crear objetos

- su único objetivo es que sirvan de superclases a las clases “reales”

Ejemplos:

- nunca crearemos objetos de la clase **Figura**
 - lo haremos de sus subclases **Círculo**, **Cuadrado**, ...
- nunca crearemos un **Vehículo**
 - crearemos un **Coche**, un **Barco**, un **Avión**, ...

La razón es que no existen “figuras” o “vehículos” genéricos

- ambos conceptos son abstracciones de los objetos reales, tales como círculos, cuadrados, coches o aviones
- a ese tipo de clases las denominaremos ***clases abstractas***

Métodos abstractos

Una clase abstracta puede tener **métodos abstractos**

- se trata de métodos sin cuerpo
- que **es obligatorio redefinir** en las subclases no abstractas

Permiten declarar en la superclase un comportamiento que deberán verificar todas sus subclases

- pero sin decir nada sobre su implementación

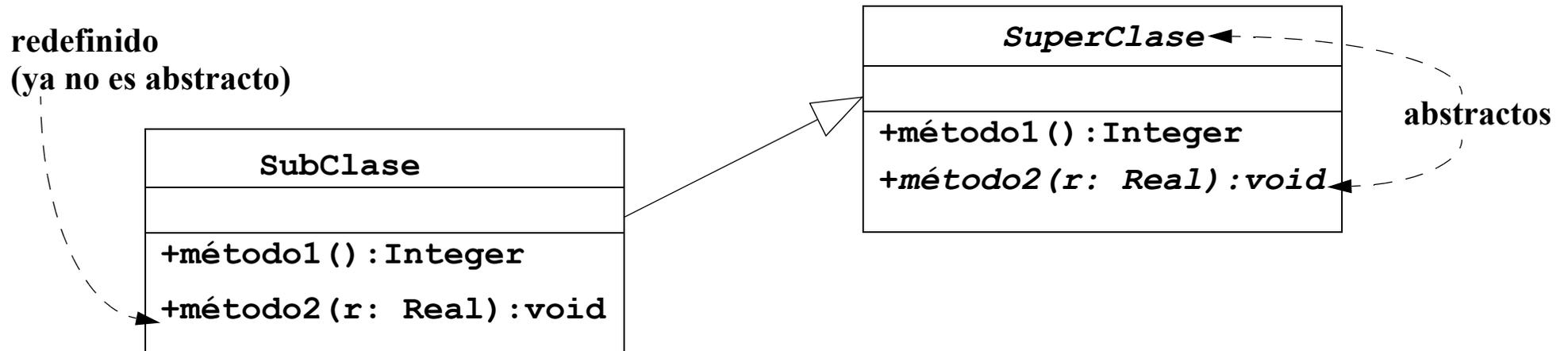
Ejemplo de método abstracto en Java

```
public abstract int métodoAbstracto(double d);
```

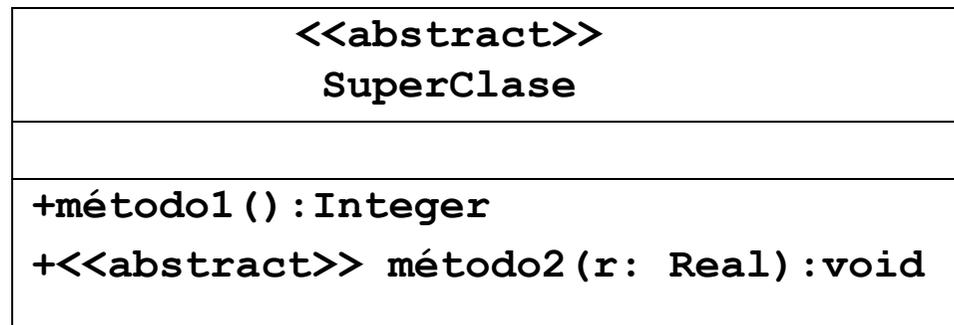
no tiene cuerpo 

Clases abstractas en diagramas de clases

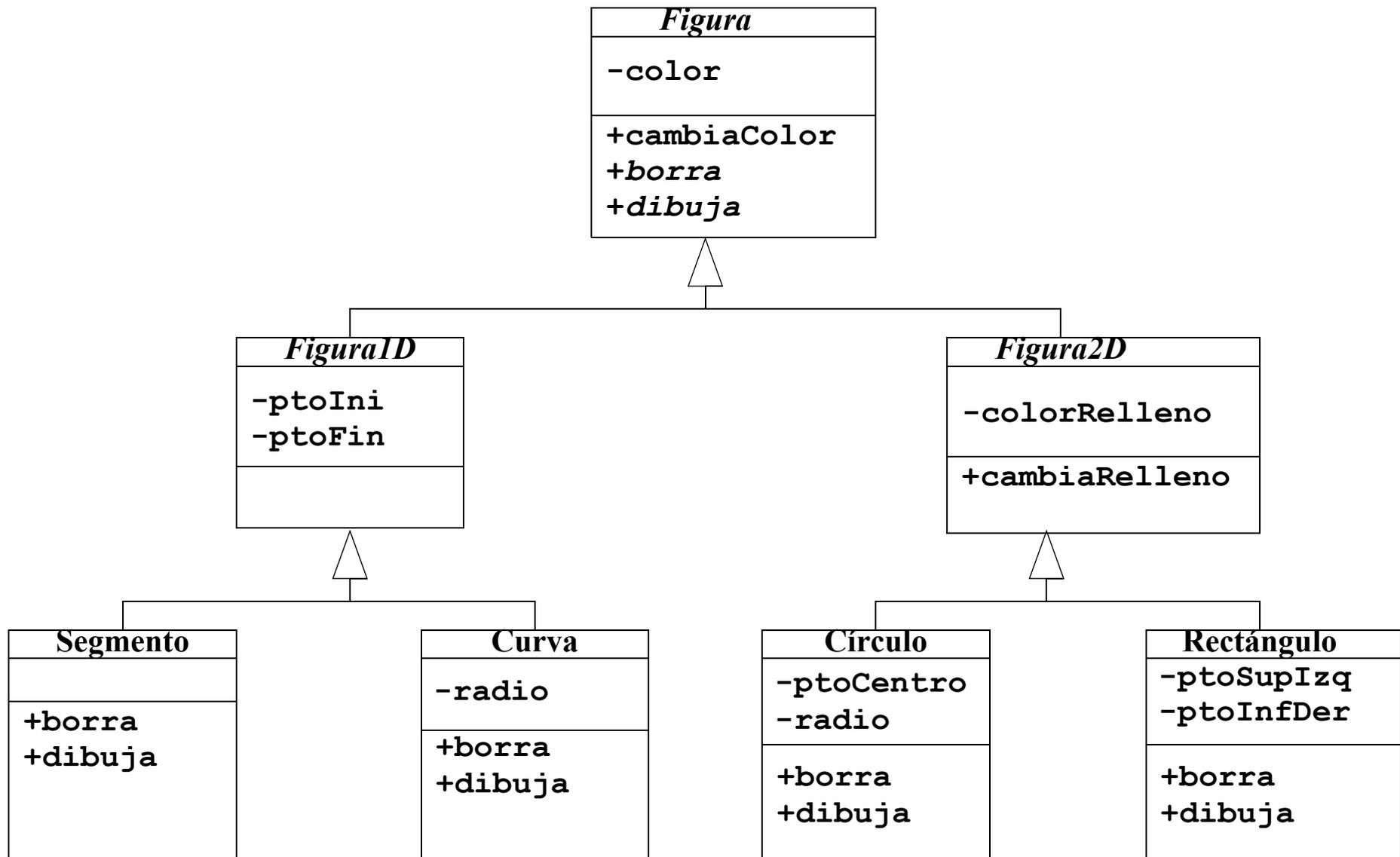
Las clases y los métodos abstractos se escriben en cursiva



También puede utilizarse el estereotipo <<abstract>>



Ejemplo: jerarquía de clases



Clases abstractas en Java

Las clases abstractas en Java se identifican mediante la palabra reservada `abstract`

```
public abstract class Figura {  
    ...  
}
```

Es un error tratar de crear un objeto de una clase abstracta

```
Figura f = new Figura(...);
```

← ERROR detectado por el compilador

Pero NO es un error utilizar referencias a clases abstractas

- que pueden apuntar a objetos de cualquiera de sus subclases (hablaremos más de ello cuando veamos el polimorfismo)

```
Figura f1 = new Círculo(...); // correcto  
Figura f2 = new Cuadrado(...); // correcto
```

Ejemplo: jerarquía de figuras

```
public abstract class Figura {
    // color del borde de la figura
    private int color;
    /** Constructor ... */
    public Figura(int color) {
        this.color=color;
    }
    /** cambia el color del borde de la figura ... */
    public void cambiaColor(int color) {
        this.color=color;
    }
    /** borra la figura (abstracto) ... */
    public abstract void borra();
    /** dibuja la figura (abstracto) ... */
    public abstract void dibuja();
}
```

```
public abstract class Figura1D extends Figura {  
  
    // puntos de comienzo y final de la figura  
    private Punto ptoIni, ptoFin;  
  
    /** Constructor ... */  
    public Figura1D(int color, Punto ptoIni,  
                   Punto ptoFin) {  
        super(color);  
        this.ptoIni = ptoIni;  
        this.ptoFin = ptoFin;)  
    }  
  
    // NO redefine ningún método abstracto  
}
```

```
public abstract class Figura2D extends Figura {  
  
    // color de relleno de la figura  
    private int colorRelleno;  
  
    /** Constructor ... */  
    public Figura2D(int color, int colorRelleno) {  
        super(color);  
        this.colorRelleno=colorRelleno;  
    }  
  
    /** cambia el color de relleno ... */  
    public void cambiaRelleno(int color) {  
        colorRelleno=color;  
    }  
  
    // NO redefine ningún método abstracto  
}
```

```
public class Recta extends Figura1D {  
  
    /** Constructor ... */  
    public Recta(int color,  
                Punto ptoIni, Punto ptoFin) {  
        super(color, ptoIni, ptoFin);  
    }  
  
    /** implementa el método abstracto borra ... */  
    @Override  
    public void borra() { implementación...; }  
  
    /** implementa el método abstracto dibuja ... */  
    @Override  
    public void dibuja() { implementación...; }  
        ....;  
    }  
}
```

```
public class Círculo extends Figura2D {
    private Punto ptoCentro;
    private double radio;
    /** Constructor ... */
    public Círculo(int color, int colorRelleno,
                  Punto ptoCentro, double radio){
        super(color, colorRelleno);
        this.ptoCentro = ptoCentro;
        this.radio = radio;
    }
    /** implementa el método abstracto borra ... */
    @Override
    public void borra() { implementación...; }
    /** implementa el método abstracto dibuja ... */
    @Override
    public void dibuja() { implementación...; }
}
```

8.3 Polimorfismo

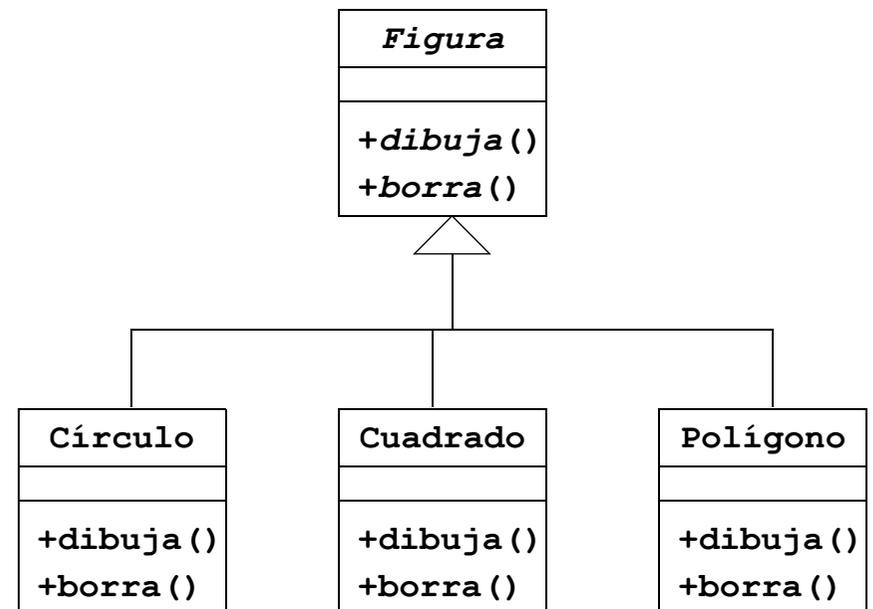
Las **operaciones polimórficas** son aquellas que hacen funciones similares con objetos diferentes

Ejemplo:

- suponer que existe la clase **Figura** y sus subclases
 - **Círculo**
 - **Cuadrado**
 - **Polígono**

Todas ellas con las operaciones:

- **dibuja()**
- **borra()**



Nos gustaría poder hacer la **operación polimórfica** `mueveFigura` que opere correctamente con cualquier clase de figura:

```
mueveFigura
  borra
  dibuja en la nueva posición
```

Esta operación polimórfica debería:

- llamar a las operaciones `borra` y `dibuja` del `Círculo` cuando la figura sea un círculo
- llamar a las operaciones `borra` y `dibuja` del `Cuadrado` cuando la figura sea un cuadrado
- etc.

Polimorfismo en Java

El polimorfismo en Java consiste en dos propiedades:

1. Una referencia a una superclase puede apuntar a un objeto de cualquiera de sus subclases

```
Vehículo v1=new Coche(Vehiculo.rojo,12345,2000);  
Vehículo v2=new Barco(Vehiculo.azul,2345);
```

2. La operación se selecciona en base a la clase del objeto, no a la de la referencia

`v1.toString()` ← usa el método de la clase `Coche`, puesto que `v1` es un coche

`v2.toString()` ← usa el método de la clase `Barco`, puesto que `v2` es un barco

Gracias a esas dos propiedades, el método `moverFigura` sería:

```
public void mueveFigura(Figura f, Posición pos){  
    f.borra();  
    f.dibuja(pos);  
}
```

Y podría invocarse de la forma siguiente:

```
Círculo c = new Círculo(...);  
Polígono p = new Polígono(...);  
mueveFigura(c, pos);  
mueveFigura(p, pos);
```

- Gracias a la primera propiedad el parámetro `f` puede referirse a cualquier subclase de `Figura`
- Gracias a la segunda propiedad en `mueveFigura` se llama a las operaciones `borra` y `dibuja` apropiadas

El lenguaje permite que una referencia a una superclase pueda apuntar a un objeto de cualquiera de sus subclases

- pero no al revés

```
Vehículo v = new Coche(...); // permitido  
Coche c = new Vehículo(...); // ¡NO permitido!
```

Justificación:

- un coche es un vehículo
 - cualquier operación de la clase `Vehículo` existe (sobrescrita o no) en la clase `Coche`
`v.operaciónDeVehículo(...); // siempre correcto`
- un vehículo no es un coche
 - sería un error tratar de invocar la operación:
`c.cilindrada(); // ERROR: cilindrada() no
// existe para un vehículo`
 - por esa razón el lenguaje lo prohíbe

Conversión de referencias (*casting*)

Es posible convertir referencias

```
Vehículo v=new Coche(...);  
Coche c=(Coche)v;
```

El *casting* **cambia el “punto de vista”** con el que vemos al objeto

- a través de **v** le vemos como un **Vehículo** (y por tanto sólo podemos invocar métodos definidos en la clase **Vehículo**)
- a través de **c** le vemos como un **Coche** (y podemos invocar cualquiera de los métodos de esa clase y de sus superclases)

Hacer una **conversión de tipos incorrecta** produce una excepción **ClassCastException** en tiempo de ejecución

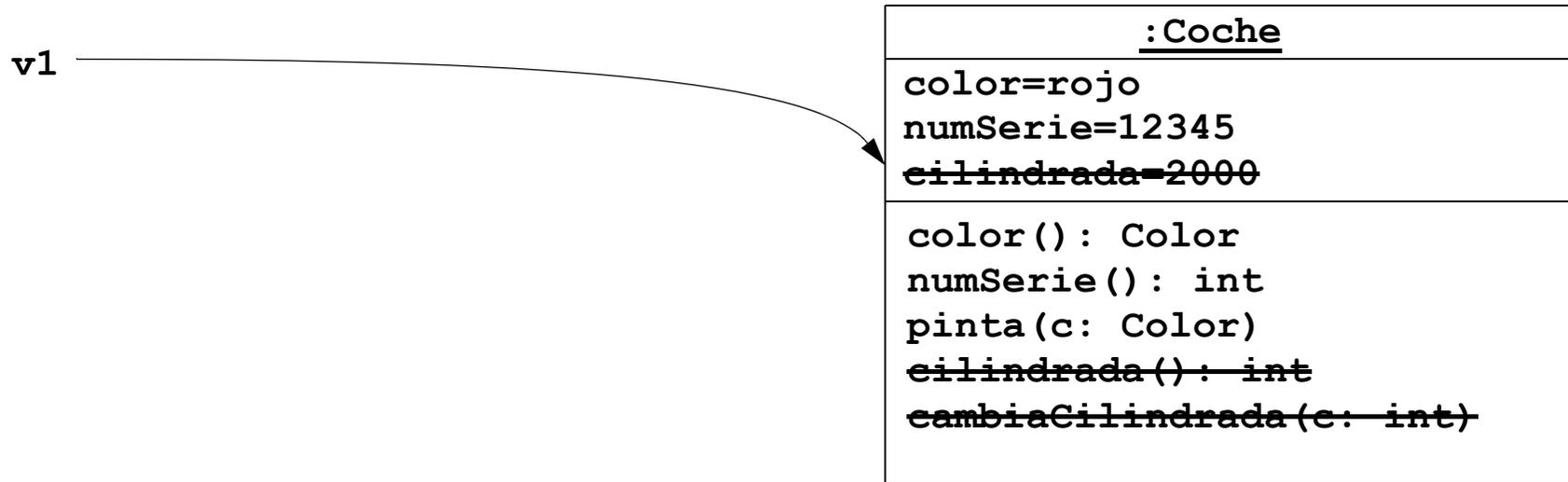
```
Vehículo v=new Vehículo(...);  
Coche c=(Coche)v;
```

lanza **ClassCastException** en tiempo de ejecuc



Cambio de “punto de vista”

```
Vehículo v1=new Coche(Vehículo.rojo,12345,2000);
```



Desde una referencia de tipo `Vehículo`, un coche se ve desde el punto de vista de un `Vehículo`

- Sólo se puede acceder a los atributos y métodos definidos en la clase `Vehículo`.

Desde una referencia de tipo `Coche` se podrían acceder a todos sus atributos y métodos.

Operador instanceof

Java proporciona el operador `instanceof` que permite conocer la clase de un objeto

```
if (v instanceof Coche) {  
    Coche c=(Coche)v;  
    ...  
}
```

NUNCA lanza `ClassCastException` (por seguro que `V` es un coche)

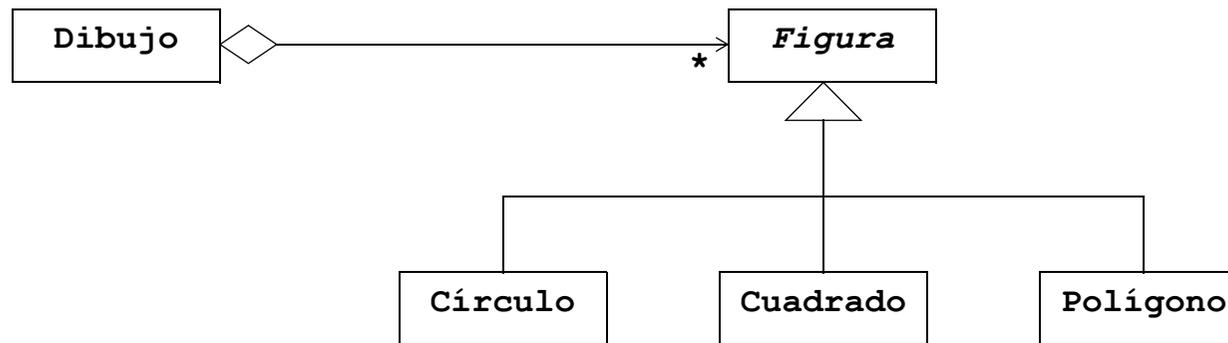
- "`v instanceof Coche`" retorna `true` si `v` apunta a un objeto de la clase `Coche` o de cualquiera de sus (posibles) subclasses

Un **uso excesivo** del operador `instanceof` :

- **Elimina las ventajas** del polimorfismo
- Revela un **diseño incorrecto** de las jerarquías de clases

Arrays de objetos de la misma jerarquía de clases

Gracias al polimorfismo es posible que un array (o un ArrayList) contenga referencias a objetos de **distintas clases de una jerarquía**



La clase **Dibujo** contiene una colección de figuras (círculos, cuadrados y polígonos)

Si creamos la clase **Triángulo** que extiende a **Figura**

- la clase **Dibujo** **no necesita ser modificada**

Ejemplo sencillo: array de figuras

```
Figura[] figuras = new Figura[3];  
figuras[0] = new Círculo(...);  
figuras[1] = new Cuadrado(...);  
figuras[2] = new Polígono(...);
```

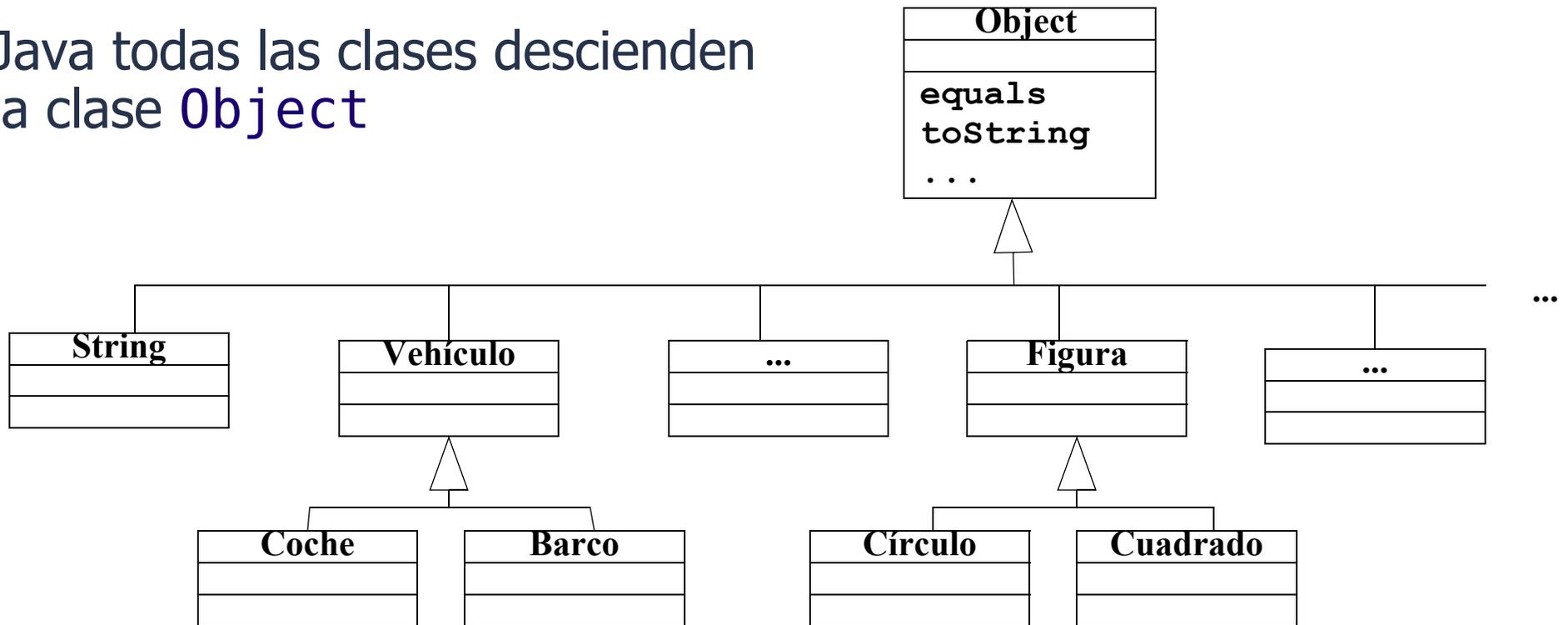
...

```
// borra todas las figuras  
for(int i=0; i<figuras.length; i++){  
    figuras[i].borra();  
}
```

Llama a la operación **borra** correspondiente a la clase del objeto

8.4 La clase Object

En Java todas las clases descienden de la clase `Object`



Es como si el compilador añadiera “**extends Object**” a todas las clases de primer nivel

```
public class Clase {...}
```

es transformado por el compilador en

```
public class Clase extends Object {...}
```

Método equals

Se encuentra definido en la clase `Object` como:

```
public class Object {  
    ...  
    public boolean equals(Object obj) {  
        return this == obj;  
    }  
    ...  
}
```

- es decir, ***compara referencias***, no contenidos

Como cualquier otro método de una superclase

- ***se puede redefinir en sus subclases***

Con lo que sabemos ahora ya podemos entender la redefinición del método `equals` para la clase `Coordenada` (vista en el tema 2):

```
public class Coordenada {
    private int x; // coordenada en el eje x
    private int y; // coordenada en el eje y
```

...

aconsejable cuando se
redefine un método para
detectar errores

Redefine el método
`equals` de la
clase `Object`

@Override

```
public boolean equals(Object obj) {
```

```
    Coordenada c = (Coordenada) obj;
```

```
    return c.x == x && c.y == y;
```

```
}
```

```
}
```

Cambio de “punto de vista”
para poder acceder a los
campos `x` e `y` de `obj`

Para ser más correctos, la redefinición del método debería ser:

```
@Override
public boolean equals(Object obj) {
    if (!(obj instanceof Coordenada))
        return false;
    Coordenada c = (Coordenada) obj;
    return c.x == x && c.y == y;
}
```

Si `obj` no es de la clase `Coordenada` (o de una de sus subclases) retorna `false` directamente y evita la excepción `ClassCastException`

O alternativamente:

```
@Override
public boolean equals(Object obj) {
    if (this.getClass() != obj.getClass())
        return false;
    Coordenada c = (Coordenada) obj;
    return c.x == x && c.y == y;
}
```

Muchas clases estándar Java (p.e. `ArrayList`) utilizan el método `equals` de la clase `Object` para comparar objetos

Por esa razón es importante que nuestras clases redefinan este método en lugar de definir uno similar

¡Incorrecto!



```
public boolean equals(Coordenada obj) {  
    // ¡NO redefina el método equals  
    // de la clase Object!  
    ...  
}
```

Método equals y la clase ArrayList

Varios métodos de la clase ArrayList utilizan el método equals del elemento para realizar búsquedas

Descripción	Interfaz
Busca el primer elemento de la lista igual a <code>ele</code> y lo elimina. Retorna <code>true</code> si ha eliminado el elemento	<code>boolean remove(E ele)</code>
Retorna <code>true</code> si la lista contiene el elemento <code>ele</code> al menos una vez	<code>boolean contains(Object ele)</code>
Retorna el índice de la primera aparición de <code>ele</code> , o <code>-1</code> si no existe	<code>int indexOf(Object ele)</code>
Retorna el índice de la última aparición de <code>ele</code> , o <code>-1</code> si no existe	<code>int lastIndexof(Object ele)</code>

Método toString

Se encuentra definido en la clase `Object` como:

```
public class Object {  
    ...  
    public String toString() {  
        return ...;  
    }  
    ...  
}
```

- es utilizado por el sistema cuando se concatena un objeto con un string, por ejemplo:
`println("Valor coordenada:" + c);`
- por defecto retorna un string con el nombre de la clase y la dirección de memoria que ocupa el objeto
`Coordenada@a34f5bd`

Una redefinición útil del método `toString` para la clase `Coordenada` podría ser:

```
@Override  
public String toString() {  
    return "(" + x + "," + y + " )";  
}
```

Con esta redefinición el segmento de código

```
Coordenada c = new Coordenada(1,2);  
System.out.println("Coord: " + c);
```

- produciría la salida por consola:

```
Coord: (1,2)
```

Método toString de las *Java Collections*

Las ***Java Collections*** (ArrayList, LinkedList, HashMap, ...) redefinen el método toString de Object

- el método toString muestra la colección con el formato [elemento 0, elemento 1, ..., elemento n-1]
- a su vez cada elemento se muestra invocando su propio método toString

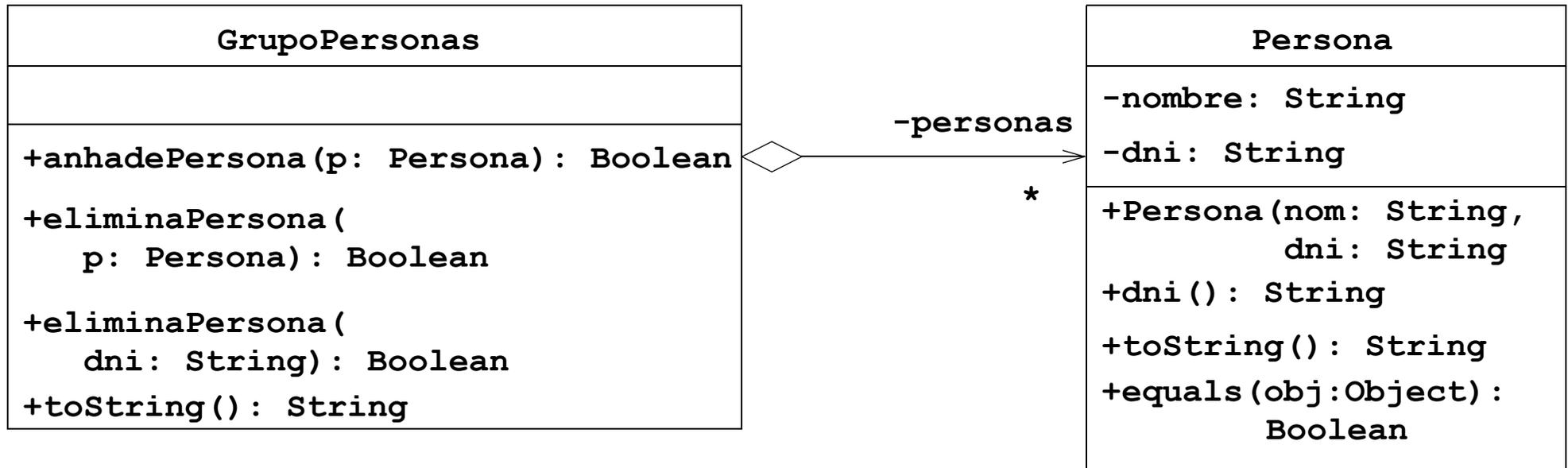
También existe un método toString para los ***arrays***

- Definido en `java.util.Arrays`:

```
int[] a = {4, 30, -5, 0, 23, -7};  
System.out.println("a:" + Arrays.toString(a));  
// muestra "a:[4, 30, -5, 0, 23, -7]"
```

- En `java.util.Arrays` existen más métodos para trabajar con arrays: `sort`, `fill`, `copyOf`, `equals`, ...

Ejemplo ArrayList, equals y toString



```
/**
 * Persona (clase muy sencilla para este ejemplo)
 * @author Métodos de Programación (UC)
 * @version curso 14-15
 */
public class Persona {

    // datos: nombre y dni
    private String nombre;
    private String dni; // único para cada persona
```

```
/**
 * Construye una persona con los datos indicados
 */
public Persona(String nombre, String dni) {
    this.nombre = nombre;
    this.dni = dni;
}
/**
 * Retorna el dni de la persona
 */
public String dni() {
    return dni;
}
@Override
public String toString() {
    return "(dni=" + dni + ", nombre=" + nombre + ")";
}
@Override
public boolean equals(Object obj) {
    if (!(obj instanceof Persona)) {
        return false;
    }
    return dni.equals(((Persona) obj).dni);
}
}
```

```
import java.util.ArrayList;

/**
 * Un grupo de personas
 * @author MP
 * @version curso 13-14
 */
public class GrupoPersonas {

    // personas pertenecientes al conjunto
    private ArrayList<Persona> personas = new ArrayList<Persona>();

    /**
     * Añade una persona al grupo (siempre que no esté ya en él)
     * @param p persona a añadir
     * @return true si se ha añadido o false en caso contrario
     */
    public boolean anhadepersona(Persona p) {
        if (personas.contains(p)) {
            return false; // la persona ya está en el grupo
        }

        // añade la persona
        personas.add(p);
        return true;
    }
}
```

```
/**
 * Elimina una persona del grupo.
 * @param p persona a eliminar
 * @return true si la persona se encontraba en el grupo
 * (y ha sido eliminada) o false en caso contrario
 */
public boolean eliminaPersona(Persona p) {
    return personas.remove(p);
}
/**
 * Elimina del grupo la persona con el DNI indicado
 * @param dni DNI de la persona a eliminar
 * @return true si la persona con ese DNI se encontraba en el
 * grupo (y ha sido eliminada) o false en caso contrario
 */
public boolean eliminaPersona(String dni) {
    for(int i=0; i<personas.size(); i++) {
        if (personas.get(i).dni().equals(dni)) {
            // encontrada la persona a eliminar
            personas.remove(i);
            return true;
        }
    }
    return false; // no encontrado
}
```

```
@Override  
public String toString() {  
    return personas.toString();  
}  
}
```